Basics of Deductive Program Verification

Claude Marché

Cours MPRI 2-36-1 “Preuve de Programme”

December 7th, 2021

Preliminaries

Very first question

Lectures in English or in French?

- Schedule on the Web page `https://marche.gitlabpages.inria.fr/lecture-deductive-verif/`
- Lectures 1,2,3,4: Claude Marché
- Lectures 5,6,7,8: Jean-Marie Madiot
- Evaluation:
  - project P using the Why3 tool (`http://why3.lri.fr`)
  - final exam E: date to decide
  - final mark = $(2E + P + max(E, P))/4$
- Project:
  - provided at the beginning of January
  - No lecture on February 1st, replaced by practical lab session (support for project)
  - due date around mid-February
- Internships (*stages*)

Outline

General Objectives

Ultimate Goal

Verify that software is free of bugs

Famous software failures:
`http://www.cs.tau.ac.il/~nachumd/horror.html`

This lecture

Computer-assisted approaches for verifying that
a software conforms to a specification

Some general approaches to Verification

Static analysis, Algorithmic Verification

- *model checking* (automata-based models)
- *abstract interpretation* (domain-specific model, e.g. numerical)

Deductive verification

- formal models using expressive logics
- verification = computer-assisted mathematical proof

Some general approaches to Verification

Refinement

- Formal models
- Code derived from model, correct by construction

A long time before success

Computer-assisted verification is an old idea

- Turing, 1948
- Floyd-Hoare logic, 1969

Success in practice: only from the mid-1990s

- Importance of the *increase of performance of computers*

A first success story:

- Paris metro line 14, using *Atelier B* (1998, refinement approach)

Other Famous Success Stories

- Flight control software of A380: *Astree* verifies absence of run-time errors (2005, abstract interpretation)
  `http://www.astree.ens.fr/`
- Microsoft's hypervisor: using Microsoft's *VCC* and the *Z3* automated prover (2008, deductive verification)
  `http://research.microsoft.com/en-us/projects/vcc/`
  More recently: verification of PikeOS
- Certified C compiler, developed using the *Coq* proof assistant (2009, correct-by-construction code generated by a proof assistant)
  `http://compcert.inria.fr/`
- L4.verified micro-kernel, using tools on top of *Isabelle/HOL* proof assistant (2010, Haskell prototype, C code, proof assistant)
  `http://www.ertos.nicta.com.au/research/l4.verified/`

Other Success Stories at Industry

- Frama-C
  - EDF: abstract interpretation
  - Airbus: deductive verification
- Spark/Ada: Verification of Ada programs
  `https://www.adacore.com/industries`

Remark

The two above use Why3 internally

Outline

Introduction, Short History

Preliminary on Automated Deduction
- Classical Propositional Logic
- First-order logic
- Logic Theories
- Limitations of Automatic Provers

Introduction to Deductive Verification
- Formal contracts
- Hoare Logic
- Dijkstra's Weakest Preconditions

Exercises

Proposition logic in a nutshell

- Syntax:

$$
\begin{array}{lcll}
\varphi & ::= & \bot \mid \top \mid A, B & \text{(atoms)} \\
 & \mid & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi & \\
 & \mid & \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi &
\end{array}
$$

- Semantics, models: truth tables

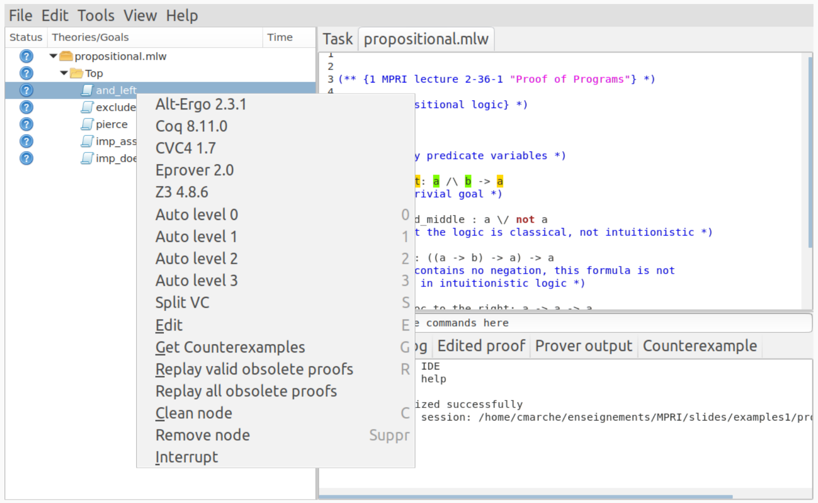
ϕ is satisfiable if it has a model
ϕ is valid if true in all models
(equivalently $\neg\phi$ is not satisfiable)

SAT is *decidable* $\rightsquigarrow$ SAT solvers

Demo with Why3

`$ why3 ide propositional.mlw`
Notice that Why3 indeed queries solvers for satisfiability of $\neg\phi$

Focus on the "Tools" menu of Why3



First-order logic in a nutshell

- Syntax:

$$
\begin{array}{rcll}
\varphi & ::= & \cdots & \\
 & | & P(t,\ldots,t) & \text{(predicates)} \\
 & | & \forall x.\ \phi \mid \exists x.\ \phi & \\
t & ::= & x & \text{variables} \\
 & | & f(t,\ldots,t) & \text{(function symbols)}
\end{array}
$$

- Semantics: models must interpret variables. C
- Satisfiability *undecidable*, but still *semi-decidable*: there exists complete systems of deduction rules (sequent calculus, natural deduction, superposition calculus)
- Examples of solvers: E, Spass, Vampire

  Implement *refutationally complete* procedure:
  if they answer 'unsat' then formula is unsatisfiable

Demo with Why3

`first-order.mlw`

Notice that Why3 logic is *typed*, and application is curryied

Logic Theories

- *Theory* = set of formulas (called *theorems*) closed by logical consequence
- *Axiomatic Theory* = set of formulas generated by axioms (or axiom schemas)
- *Consistent Theory*

  for no P, P and $\neg P$ are both theorems
  equivalently: 'false' is not a theorem
  equivalently: the theory has models
- *Consistent Axiomatization*
  'false' is not derivable

Theory of Equality

$$\forall x.\ x = x$$

$$\forall x, y.\ x = y \rightarrow y = x$$

$$\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$$

(congruence) for all function symbols f of arity k:

$$\forall x_1, y_1 \ldots, x_k, y_k.\ x_1 = y_1 \wedge \cdots \wedge x_k = y_k \rightarrow f(x_1,\ldots,x_k) = f(y_1,\ldots,y_k)$$

and for all predicates p of arity k:

$$\forall x_1, y_1 \ldots, x_k, y_k.\ x_1 = y_1 \wedge \cdots \wedge x_k = y_k \rightarrow p(x_1,\ldots,x_k) \rightarrow p(y_1,\ldots,y_k)$$

Theory of Equality, Continued

$$\forall x.\ x = x$$

$$\forall x, y.\ x = y \rightarrow y = x$$

$$\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$$

(congruence) ...

- General first-order deduction bad in such a case $\rightsquigarrow$ dedicated methods
  - paramodulation
  - congruence closure (for quantifier-free fragment)
- SMT solvers (Alt-Ergo, CVC4, Z3) implement dedicated (semi-)decision procedures

Demo with Why3

`equality.mlw`

Theories Continued

Theory of a given model

= formulas true in this model

- Central example: theory of linear integer arithmetic, i.e. formulas using $+$ and $\leq$
  - First-order theory is known to be decidable (Presburger)
  - SMT solvers typically implement a procedure for the existential fragment
- Also: theory of (non-linear) real arithmetic is decidable (Tarski)

Non-linear Integer Arithmetic

(a.k.a. Peano Arithmetic)

First-Order Integer Arithmetic

All valid first-order formulas on integers with $+$, $\times$ and $\leq$

- This theory is not even semi-decidable
- SMT solvers implement incomplete satisfiability checks: if solver answers 'unsat' then it is unsatisfiable

Demo with Why3

`arith.mlw`

Digression about Non-linear Integer Arithmetic

Representation Theorem (Gödel)

Every recursive function f is representable by a predicate ϕ_f such that

$$\phi_f(x_1, \ldots, x_k, y)$$

is true if and only if

$$y = f(x_1, \ldots, x_k)$$

First incompleteness Theorem (Gödel)

That theory is not recursively axiomatizable

Summary of prover limitations

- Superposition solvers (E, Spass,)
  - do not support well theories except equality
  - do quite well with quantifiers
- SMT solvers (Alt-Ergo, CVC4, Z3)
  - several theories are built-in
  - weaker with quantifiers
- None support reasoning by induction

Outline

IMP language

IMP language

A very basic imperative programming language

- only global variables
- only integer values for expressions
- basic statements:
  - assignment x `<-` e
  - sequence s_1 ; s_2
  - conditionals `if` e `then` s_1 `else` s_2
  - loops `while` e `do` s
  - no-op `skip`

Formal Contracts

General form of a program:

Contract

- *precondition*: expresses what is assumed before running the program
- *post-condition*: expresses what is supposed to hold when program exits

Demo with Why3

`contracts.mlw`

Hoare triples

- *Hoare triple* : notation $\{P\}s\{Q\}$
- P : formula called the *precondition*
- Q : formula called the *postcondition*

Intended meaning

$\{P\}s\{Q\}$ is true if and only if:
when the program s is executed in any state satisfying P, then (if execution terminates) its resulting state satisfies Q

This is a *Partial Correctness*: we say nothing if s does not terminate

Examples

Examples of valid triples for partial correctness:

- $\{x = 1\} x \texttt{<-} x + 2 \{x = 3\}$
- $\{x = y\} x \texttt{<-} x + y \{x = 2 * y\}$
- $\{\exists v.\ x = 4 * v\} x \texttt{<-} x + 42 \{\exists w.\ x = 2 * w\}$
- $\{\mathit{true}\}$`while 1 do skip`$\{\mathit{false}\}$

Running Example

Three global variables `n`, `count`, and `sum`

```
count <- 0; sum <- 1;
while sum <= n do
  count <- count + 1; sum <- sum + 2 * count + 1
```

What does this program compute?

(assuming input is `n` and output is `count`)

Informal specification:

- at the end of execution of this program, `count` contains the square root of `n`, rounded downward
- e.g. for `n=42`, the final value of `count` is 6.

See file `imp_isqrt.mlw`

Hoare logic as an Axiomatic Semantics

Original Hoare logic *[∼ 1970]*

Axiomatic Semantics of programs

Set of *inference rules* producing triples

$$\frac{}{\{P\}\texttt{skip}\{P\}}$$

$$\frac{}{\{P[x \leftarrow e]\} x \texttt{<-} e \{P\}}$$

$$\frac{\{P\} s_1 \{Q\} \qquad \{Q\} s_2 \{R\}}{\{P\} s_1 ; s_2 \{R\}}$$

- Notation $P[x \leftarrow e]$: replace all occurrences of program variable x by e in P.

Hoare Logic, continued

Frame rule:

$$\frac{\{P\} s \{Q\}}{\{P \wedge R\} s \{Q \wedge R\}}$$

with R a formula where no variables assigned in s occur

Consequence rule:

$$\frac{\{P'\} s \{Q'\} \qquad \models P \rightarrow P' \qquad \models Q' \rightarrow Q}{\{P\} s \{Q\}}$$

- Example: proof of

$$\{x = 1\} x \texttt{<-} x + 2 \{x = 3\}$$

Proof of the example

$$\dfrac{\dfrac{}{\{x+2=3\}x \leftarrow x+2\{x=3\}} \qquad \begin{array}{l}\models x=1 \rightarrow x+2=3 \\ \models x=3 \rightarrow x=3\end{array}}{\{x=1\}x \leftarrow x+2\{x=3\}}$$

Hoare Logic, continued

Rules for if and while :

$$\frac{\{P \wedge e\}s_1\{Q\} \qquad \{P \wedge \neg e\}s_2\{Q\}}{\{P\}\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2\{Q\}}$$

$$\frac{\{I \wedge e\}s\{I\}}{\{I\}\texttt{while } e \texttt{ do } s\{I \wedge \neg e\}}$$

I is called a *loop invariant*

Informal justification of the while rule

$$\frac{\{I \wedge e\}s\{I\}}{\{I\}\texttt{while } e \texttt{ do } s\{I \wedge \neg e\}}$$

I	invariant initially valid
$I \wedge e$	condition assumed true
s	execution of loop body
I	invariant re-established
$I \wedge e$	condition assumed true
s	execution of loop body
I	invariant re-established
$\vdots$	any number of iterations
I	invariant re-established
$I \wedge \neg e$	loop exits when condition false

Example: isqrt(42)

Exercise: prove of the triple

$$\{n \geq 0\}\ ISQRT\ \{count^2 \leq n \wedge n < (count+1)^2\}$$

Could we do that by hand?

Back to demo: file `imp_isqrt.mlw`

Warning

Finding an adequate loop invariant is a major difficulty

Beyond Axiomatic Semantics

- Operational Semantics
- Semantic Validity of Hoare Triples
- Hoare logic as correct deduction rules

Operational semantics

[Plotkin 1981, structural operational semantics (SOS)]

- we use a standard *small-step semantics*
- *program state*: describes content of global variables at a given time. It is a finite map Σ associating to each variable x its current value denoted $\Sigma(x)$.
- Value of an expression e in some state Σ:
  - denoted $[\![e]\!]_\Sigma$
  - always defined, by the following recursive equations:

$$\begin{array}{rcl} [\![n]\!]_\Sigma & = & n \\ [\![x]\!]_\Sigma & = & \Sigma(x) \\ [\![e_1 \ op \ e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \ [\![op]\!] \ [\![e_2]\!]_\Sigma \end{array}$$

  - $[\![op]\!]$ natural semantic of operator op on integers (with relational operators returning 0 for false and $\neq 0$ for true).

Semantics of statements

Semantics of statements: defined by judgment

$$\Sigma, s \quad \rightsquigarrow \quad \Sigma', s'$$

meaning: in state Σ, executing one step of statement s leads to the state Σ' and the remaining statement to execute is s'.
The semantics is defined by the following rules.

$$\frac{}{\Sigma, x \mathrel{<\!\!-} e \rightsquigarrow \Sigma\{x \leftarrow [\![e]\!]_\Sigma\}, \mathtt{skip}}$$

$$\frac{\Sigma, s_1 \rightsquigarrow \Sigma', s_1'}{\Sigma, (s_1; s_2) \rightsquigarrow \Sigma', (s_1'; s_2)}$$

$$\frac{}{\Sigma, (\mathtt{skip}; s) \rightsquigarrow \Sigma, s}$$

Semantics of statements, continued

$$\frac{[\![e]\!]_\Sigma \neq 0}{\Sigma, \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \rightsquigarrow \Sigma, s_1}$$

$$\frac{[\![e]\!]_\Sigma = 0}{\Sigma, \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \rightsquigarrow \Sigma, s_2}$$

$$\frac{[\![e]\!]_\Sigma \neq 0}{\Sigma, \mathtt{while}\ e\ \mathtt{do}\ s \rightsquigarrow \Sigma, (s; \mathtt{while}\ e\ \mathtt{do}\ s)}$$

$$\frac{[\![e]\!]_\Sigma = 0}{\Sigma, \mathtt{while}\ e\ \mathtt{do}\ s \rightsquigarrow \Sigma, \mathtt{skip}}$$

Execution of programs

- $\rightsquigarrow$: a binary relation over pairs (state,statement)
- transitive closure : $\rightsquigarrow^+$
- reflexive-transitive closure : $\rightsquigarrow^*$

In other words:

$$\Sigma, s \overset{*}{\rightsquigarrow} \Sigma', s'$$

means that statement s, in state Σ, reaches state Σ' with remaining statement s' after executing some finite number of steps.

Running example:

$$\{n = 42, count = ?, sum = ?\}, ISQRT \rightsquigarrow^* \{n = 42, count = 6, sum = 49\}, \texttt{skip}$$

Execution and termination

- any statement except `skip` can execute in any state
- the statement `skip` alone represents the final step of execution of a program
- there is no possible *runtime error*.

Definition

Execution of statement s in state Σ *terminates* if there is a state Σ' such that $\Sigma, s \rightsquigarrow^* \Sigma', \texttt{skip}$

- since there are no possible runtime errors, s does not terminate means that s *diverges* (i.e. executes infinitely).

Semantics of formulas

$[\![p]\!]_\Sigma$:

- semantics of formula p in program state Σ
- is a logic formula where no program variables appear anymore
- defined recursively as follows.

$$\begin{array}{rcl} [\![e]\!]_\Sigma & = & [\![e]\!]_\Sigma \neq 0 \\ [\![p_1 \wedge p_2]\!]_\Sigma & = & [\![p_1]\!]_\Sigma \wedge [\![p_2]\!]_\Sigma \\ & \vdots & \end{array}$$

where semantics of expressions is augmented with

$$\begin{array}{rcl} [\![v]\!]_\Sigma & = & v \\ [\![x]\!]_\Sigma & = & \Sigma(x) \end{array}$$

Notations:

- $\Sigma \models p$: the formula $[\![p]\!]_\Sigma$ is *valid*
- $\models p$: formula $[\![p]\!]_\Sigma$ holds in all states Σ.

Semantics of formulas

Other presentation of the semantics: $[\![p]\!]_\Sigma$:

- inline semantic of first-order formula
- $[\![e]\!]_{\Sigma,\mathcal{V}}$ with $\mathcal{V}$ mapping of logic variables to integers.
- defined recursively as follows.

$$\begin{array}{rcl} [\![p_1 \wedge p_2]\!]_{\Sigma,\mathcal{V}} & = & \begin{cases} \top & \text{if } [\![p_1]\!]_{\Sigma,\mathcal{V}} = \top \text{ and } [\![p_2]\!]_{\Sigma,\mathcal{V}} = \top \\ \bot & \end{cases} \\ [\![\forall x.e]\!] & = & \top \text{ if for all } v.\ [\![e]\!]_{\Sigma,\mathcal{V}[x \leftarrow v]} = \top \\ & \vdots & \end{array}$$

where semantics of expressions is augmented with

$$\begin{array}{rcl} [\![v]\!]_{\Sigma,\mathcal{V}} & = & \mathcal{V}(v) \\ [\![x]\!]_{\Sigma,\mathcal{V}} & = & \Sigma(x) \end{array}$$

Soundness

Definition (Partial correctness)

Hoare triple $\{P\}s\{Q\}$ is said *valid* if:
for any states Σ, Σ', if

- $\Sigma, s \rightsquigarrow^* \Sigma', \texttt{skip}$ and
- $\Sigma \models P$

then $\Sigma' \models Q$

Theorem (Soundness of Hoare logic)

The set of rules is correct: any derivable triple is valid.

This is *proved by induction on the derivation tree* of the considered triple.
For each rule: assuming that the triples in premises are valid, we show that the triple in conclusion is valid too.

Digression: Completeness of Hoare Logic

Two major difficulties for proving a program

- *guess the appropriate intermediate formulas* (for sequence, for the loop invariant)
- *prove the logical premises of consequence rule*

Theoretical question: completeness. Are all valid triples derivable from the rules?

Theorem (Relative Completeness of Hoare logic)

The set of rules of Hoare logic is relatively complete: if the logic language is expressive enough, then any valid triple $\{P\}s\{Q\}$ can be derived using the rules.

[Cook, 1978] "Expressive enough": representability of any recursive function
Yet, this does not provide an effective recipe to discover suitable loop invariants (see also the theory of abstract interpretation *[Cousot, 1990]*)

Annotated Programs

Goal

Add automation to the Hoare logic approach

We augment IMP with *explicit loop invariants*

$$\texttt{while } e \texttt{ invariant } I \texttt{ do } s$$

Weakest liberal precondition

[Dijkstra 1975]

Function $\mathrm{WLP}(s, Q)$:

- s is a statement
- Q is a formula
- returns a formula

It should return the *minimal precondition* P that validates the triple $\{P\}s\{Q\}$

Definition of WLP(s, Q)

Recursive definition:

$$
\begin{array}{rcl}
\mathrm{WLP}(\texttt{skip}, Q) & = & Q \\
\mathrm{WLP}(x \texttt{ <- } e, Q) & = & Q[x \leftarrow e] \\
\mathrm{WLP}(s_1; s_2, Q) & = & \mathrm{WLP}(s_1, \mathrm{WLP}(s_2, Q)) \\
\mathrm{WLP}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, Q) & = & \\
(e \rightarrow \mathrm{WLP}(s_1, Q)) & \wedge & (\neg e \rightarrow \mathrm{WLP}(s_2, Q))
\end{array}
$$

Definition of WLP(s, Q), continued

$$
\begin{array}{ll}
\mathrm{WLP}(\texttt{while } e \texttt{ invariant } I \texttt{ do } s, Q) = & \\
\quad I \wedge & \text{(invariant true initially)} \\
\quad \forall v_1, \ldots, v_k. & \\
\qquad (((e \wedge I) \rightarrow \mathrm{WLP}(s, I)) & \text{(invariant preserved)} \\
\qquad \wedge((\neg e \wedge I) \rightarrow Q))[w_i \leftarrow v_i] & \text{(invariant implies post)}
\end{array}
$$

where $w_1, \ldots, w_k$ is the set of assigned variables in statement s and $v_1, \ldots, v_k$ are fresh logic variables

Examples

$$\mathrm{WLP}(x \texttt{ <- } x + y, x = 2y) \equiv x + y = 2y$$

$$
\begin{array}{l}
\mathrm{WLP}(\texttt{while } y > 0 \texttt{ invariant } \mathit{even}(y) \texttt{ do } y \texttt{ <- } y - 2, \mathit{even}(y)) \equiv \\
\quad \mathit{even}(y) \wedge \\
\quad \forall v, ((v > 0 \wedge \mathit{even}(v)) \rightarrow \mathit{even}(v - 2)) \\
\qquad \wedge((v \leq 0 \wedge \mathit{even}(v)) \rightarrow \mathit{even}(v))
\end{array}
$$

Soundness

Theorem (Soundness)

For all statement s and formula Q, $\{\mathrm{WLP}(s, Q)\}s\{Q\}$ is valid.

Proof by induction on the structure of statement s.

Consequence

For proving that a triple $\{P\}s\{Q\}$ is valid, it suffices to prove the formula $P \rightarrow \mathrm{WLP}(s, Q)$.

This is basically the goal that Why3 produces

Outline

Exercise 1

Consider the following (inefficient) program for computing the sum $a + b$.

```
x <- a; y <- b;
while y > 0 do
  x <- x + 1; y <- y - 1
```

(Why3 file to fill in: `imp_sum.mlw`)

- Propose a post-condition stating that the final value of x is the sum of the values of a and b
- Find an appropriate loop invariant
- Prove the program.

Exercise 2

The following program is one of the original examples of Floyd.

```
q <- 0; r <- x;
while r >= y do
  r <- r - y; q <- q + 1
```

(Why3 file to fill in: `imp_euclide.mlw`)

- Propose a formal precondition to express that x is assumed non-negative, y is assumed positive, and a formal post-condition expressing that q and r are respectively the quotient and the remainder of the Euclidean division of x by y.
- Find appropriate loop invariants and prove the correctness of the program.

Exercise 3

Let's assume given in the underlying logic the functions div2(x) and mod2(x) which respectively return the division of x by 2 and its remainder. The following program is supposed to compute, in variable r, the power x^n.

```
r <= 1; p <- x; e <- n;
while e > 0 do
  if mod2(e) <> 0 then r <- r * p;
  p <- p * p;
  e <- div2(e);
```

(Why3 file to fill in: `power_int.mlw`)

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
- Find an appropriate loop invariant, and prove the program.

Exercise 4

The Fibonacci sequence is defined recursively by $fib(0) = 0$, $fib(1) = 1$ and $fib(n+2) = fib(n+1) + fib(n)$. The following program is supposed to compute fib in linear time, the result being stored in y.

```
y <- 0; x <- 1; i <- 0;
while i < n do
   aux <- y; y <- x; x <- x + aux; i <- i + 1
```

- Assuming fib exists in the logic, specify appropriate pre- and post-conditions.
- Prove the program.

Exercise (original Floyd rule for assignment)

1. *Prove that the triple*

$$\{P\}x \leftarrow e\{\exists v,\ e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}$$

   is valid with respect to the operational semantics.
2. *Show that the triple above can be proved using the rules of Hoare logic.*

Let us assume that we replace the standard Hoare rule for assignment by the Floyd rule

$$\frac{}{\{P\}x \leftarrow e\{\exists v,\ e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}}$$

3. *Show that the triple* $\{P[x \leftarrow e]\}x \leftarrow e\{P\}$ *can be proved with the new set of rules.*

Bibliography

Cook(1978) S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978. doi: 10.1137/0207005.

Cousot(1990) P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.

Dijkstra(1975) E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975.

Bibliography

Floyd(1967) R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

Hoare(1969) C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10):576–580 and 583, Oct. 1969.

Plotkin(2004) G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004. doi: 10.1016/j.jlap.2004.03.009.